```
import sys, getopt
from PyQt4 import QtGui, QtCore
from PyQt4.QtGui import *
from PyQt4.QtCore import *
import numpy as np
from components import *
from itertools import izip, izip_longest
import cv2
from PyQt4.QtGui import QDialog
from mapTransform import Ui_Window
from os import path
from subprocess import call

import rospy

class MainWindow(QDialog, Ui_Window):
    def __init__(self, semantic, slam, parent=None):
        super(QDialog, self).__init__(parent)
        self.setupUi(self)
        self.setWindowTitle('Main Window')

        # Sets up the maps
        self.img_1 = semantic
        self.slam_meta_data = yaml_to_meta_data(slam, "MapMetaData")
        self.slam_origin = self.slam_meta_data.origin
        self.slam_res = self.slam_meta_data.resolution
        #self.img_2 = path.dirname(slam) + "/" +    self.slam_meta_data.image
        self.img_2 = self.slam_meta_data.image
        print self.img_2
        print self.img_1
        self.map1 = DrawMap(self.img_1, self)
        self.source.setScene( self.map1 )
        self.map2 = DrawMap(self.img_2, self)
        self.destination.setScene( self.map2 )

        # Set up variables for point registration and transformation, etc
        self.src = [(-1, -1), (-1, -1), (-1, -1)]
        self.dst = [(-1, -1), (-1, -1), (-1, -1)]
        self.robot1 = DrawRobot()
        self.robot2 = DrawRobot()
        self.robot1.setVisible(False)
        self.robot2.setVisible(False)
        self.robot = RobotHandler(self.robot1, self.robot2)
        self.export_ready = False
        # disable buttons until they are ready
        self.toggleRobot.setEnabled(False)
        self.show_zones_ckbox.setEnabled(False)
        self.export_btn.setEnabled(False)
        self.apply_transform_btn.setEnabled(False)
        self.transform_btn.setEnabled(False)

        self.map1.addItem(self.robot1)
        self.map2.addItem(self.robot2)

        self.transform_btn.setToolTip("Apply Transform and show the result")
        self.transform_btn.clicked.connect(self.transform_map)

        self.export_btn.setToolTip("Save the transformed map")
        self.export_btn.clicked.connect(self.export_map)
        self.update_labels()

        self.apply_transform_btn.setToolTip("Applies Transform only")
        self.apply_transform_btn.clicked.connect(self.triangulate)

        self.clearAll_btn.setToolTip("Clears all points from  both maps")
        self.clearAll_btn.clicked.connect(self.clear_all)
        self.clearUnmatched_btn.setToolTip("Clears points which do not have a correpsonding
point in the other map")
        self.clearUnmatched_btn.clicked.connect(self.clear_unmatched)

        self.loadPoint_btn.setToolTip("Load points into the map")
        self.loadPoint_btn.clicked.connect(self.loadPoints)
        self.exportPoint_btn.setToolTip("Save matching points. Exporting will do this")
        self.exportPoint_btn.clicked.connect(self.savePoints)

        # The signals are emitted after a click in the map window
        self.map1.register.connect(self.update_labels)
        self.map2.register.connect(self.update_labels)

        # Setting up the robot toggled checkbox
        self.toggleRobot.stateChanged.connect(self.robot_toggle)
        self.show_zones_ckbox.stateChanged.connect(self.zone_toggle)

        self.exportZone_btn.clicked.connect(self.export_zones)
        self.importZone_btn.clicked.connect(self.import_zones)
        #Data goes to zoneData, which is a text field
        # added by Matt
        self.timer = rospy.Timer(rospy.Duration(1.0), self.callback)  # call the function "callback"
(below) once per second.
        # self.timer.shutdown()  # this is how you would stop the callback.

        # added by Matt
        from geometry_msgs.msg import PoseWithCovarianceStamped
        rospy.Subscriber('/amcl_pose', PoseWithCovarianceStamped, self.move_robot)

    # added by Matt
    def callback(self, timer_event):
        print('Robot positions:')

        # Semantic map robot
        pos1 = self.robot.robot_1.get_pos()
        pos1m = tuple(p*.05 for p in pos1)  # convert to meters
        print(pos1, pos1m)

        # SLAM map robot
        pos2 = self.robot.robot_2.get_pos()
        pos2m = tuple(p*.05 for p in pos2)  # convert to meters
        print(pos2, pos2m)

        print('')
        print('')

    # added by Matt
    def move_robot(self, pose_with_covariance):
        # Get the x- and y-position of the turtlebot.
        # In meters, relative to /map frame.
        x = pose_with_covariance.pose.pose.position.x
        y = pose_with_covariance.pose.pose.position.y

        #x = 0.0  # HACK: see where the origin is.
        #y = 0.0

        # Offset because the origin of the map file is at [-23.4, -34.6, 0.0]
        x -= -23.4
        y = 57.6 - (y + 34.6)  # y=0 is at the top of the image

        # Convert from meters to pixels
        x /= 0.05
        y /= 0.05
        # Update robot positions
        self.robot.robot_2.setPos(x,y)  # move SLAM robot
        self.robot.set_robot_1()


    def loadPoints(self):
        fname = QFileDialog.getOpenFileName(self, 'Open File', "", 'YAML Files (*.yaml)')
        if fname.isEmpty():
            return
        fname = str(fname)
        meta_data = yaml_to_meta_data(fname, "RegisteredMapMetaData")
        slam_nodes = path.dirname(fname) + "/" + meta_data.slam_nodes
        first_line = True
        with open(slam_nodes, 'r') as f:
            for line in f:
                if first_line:
                    # do nothing
                    first_line = False
                elif '#' not in line:
                    # This line is not a comment
                    s = line.split()
                    x = float(s[1])
                    y = float(s[2])
                    p = QPoint(x, y)
                    self.map2.select_pix(p)
        semantic_nodes = path.dirname(fname) + "/" + meta_data.semantic_nodes
        first_line = True
        with open(semantic_nodes, 'r') as f:
            for line in f:
                if first_line:
                    # do nothing
                    first_line = False
                elif '#' not in line:
                    # This line is not a commnet
                    s = line.split()
                    x = float(s[1])
                    y = float(s[2])
                    p = QPoint(x, y)
                    self.map1.select_pix(p)

    def savePoints(self):
        counter = 0
        slam = ""
        semantic = ""
        for p1, p2 in izip(self.map1.get_points(), self.map2.get_points()):
            if p1 != None and p2 != None:
                counter += 1
                semantic += str(counter) + " " + str(p1[0]) + " " + str(p1[1]) + "\n"
                slam += str(counter) + " " + str(p2[0]) + " " + str(p2[1]) + "\n"
        semantic = str(counter) + " 2 0 1\n" + semantic
        slam = str(counter) + " 2 0 1\n" + slam
        f1 = open('semantic.node', 'w')
        f2 = open('slam.node', 'w')
        f1.write(semantic)
        f2.write(slam)
        f1.close()
        f2.close()

        call(["mv", "semantic.node", "register/"])
        call(["mv", "slam.node", "register/"])

    def clear_all(self):
        for p in self.map1.points:
            if p != None:
                p.ask_to_be_deleted()
        for p in self.map2.points:
            if p != None:
                p.ask_to_be_deleted()

    def clear_unmatched(self):
        for p1, p2 in izip_longest(self.map1.points, self.map2.points):
            # print p1, p2
            if p1 == None and p2 != None:
                p2.ask_to_be_deleted()
            if p2 == None and p1 != None:
                p1.ask_to_be_deleted()

    # Updates the labels telling how many points there are
    def update_labels(self):
        map_1_pts = self.map1.get_num_points()
        map_2_pts = self.map2.get_num_points()
        self.label_4.setText(str(map_1_pts))
        self.label_5.setText(str(map_2_pts))
        if map_1_pts > 2 and map_2_pts > 2:
            self.export_btn.setEnabled(True)
            self.apply_transform_btn.setEnabled(True)
            self.transform_btn.setEnabled(True)
        else:
```

```
        self.export_btn.setEnabled(False)
        self.apply_transform_btn.setEnabled(False)
        self.transform_btn.setEnabled(False)

# Add (or remove) a robot from the scene
def robot_toggle(self):
    if self.toggleRobot.isChecked():
        self.robot.setTransforms(self.map1.get_points(), self.map2.get_points())
    self.robot.setEnabled(self.toggleRobot.isChecked())

# Toggle the viewing of the zones on the maps
def zone_toggle(self):
    if self.show_zones_ckbox.isChecked():
        self.map1_zones = []
        self.map2_zones = []
        for zone in self.myYaml['Zone List']:
            new_zone = Zone()
            new_zone.setup_from_dict(zone)
            self.map1_zones.append(new_zone)
            self.map1.addItem(new_zone)
        if self.robot.ready:
            for zone in self.myYaml['Zone List']:
                new_zone = Zone()
                pts = []
                for point in zone['Points']:
                                    print point
                    x = int(point['x'])
                    y = int(point['y'])
                                    print x,y
                    conv_pt = self.robot.convert_to_2(QPoint(x, y))
                                    print conv_pt
                    if conv_pt == None:
                        print "Transformation does not encapsulate this Zone!"
                        return
                    pts.append(conv_pt)
                new_zone.setup(int(zone['Mode']), pts)
                self.map2_zones.append(new_zone)
                self.map2.addItem(new_zone)
    else:
        for zone in self.map1_zones:
            self.map1.removeItem(zone)
        for zone in self.map2_zones:
            self.map2.removeItem(zone)
    # Construct an ordered list of nodes from the given node file
    def nodes(self, node_file):
        nodes = [None]
        first_line = True
        with open(node_file, 'r') as f:
            for line in f:
                if first_line:
                    # do nothing
                    first_line = False
                elif '#' not in line:
                    # This line is not a comment
                    s = line.split()
                    x = float(s[1])
                    y = float(s[2])
                    nodes.append((x, y))
        return nodes

    # Using matching points, view the transformation of the maps
    def transform_map(self):
        self.triangulate()

        slam_map = cv2.imread(self.img_2, 0)
        rows, cols = cv2.imread(self.img_1, 0).shape
        output = np.zeros((rows, cols), np.uint8)
        slam_nodes = self.nodes("register/slam.1.node")
        semantic_nodes = self.nodes("register/semantic.1.node")
        # From the ele file, color triangles
        first_line = True
        with open("register/slam.1.ele", 'r') as f:
            for line in f:
                if first_line:
                    # Do nothing
                    first_line = False
                elif '#' not in line:
                    # This line is not a comment
                    s = line.split()
                    node_index_1 = int(s[1])
                    node_index_2 = int(s[2])
                    node_index_3 = int(s[3])
                    slam_pts = [slam_nodes[node_index_1], slam_nodes[node_index_2],
slam_nodes[node_index_3]]
                    semantic_pts = [semantic_nodes[node_index_1], semantic_nodes[node_index_2],
semantic_nodes[node_index_3]]
```

```
                    transform = cv2.getAffineTransform(np.array(slam_pts, dtype='float32'),
np.array(semantic_pts, dtype='float32'))
                    if transform != None:
                        all_transformed = cv2.warpAffine(slam_map, transform, (cols, rows))
                        area = np.array(semantic_pts, dtype='int32')
                        area = area.reshape((-1, 1, 2))
                        mask = np.zeros((rows, cols), np.uint8)
                        cv2.fillPoly(mask, [area], 255)
                        tmp = cv2.bitwise_and(all_transformed, mask)
                        output = cv2.add(tmp, output)
            cv2.imshow('Output', output)
            cv2.waitKey(0)
            cv2.destroyAllWindows()

    # Saves the values in a yaml file in the current directory
    def export_map(self):
        self.triangulate()

        # Write the file that relates the two maps.
        yaml = "semantic_map: " + path.basename(self.img_1) + "\n"
        yaml += "slam_map: " + path.basename(self.img_2) + "\n"
        yaml += "origin: " + str(self.slam_origin) + "\n"
        yaml += "resolution: " + str(self.slam_res) + "\n"
        src = cv2.imread(self.img_1, 0)
        dst = cv2.imread(self.img_2, 0)
        src_rows, src_cols = src.shape
        dst_rows, dst_cols = dst.shape
        yaml += "slam_width: " + str(dst_cols) + "\n"
        yaml += "slam_height: " + str(dst_rows) + "\n"
        yaml += "semantic_width: " + str(src_cols) + "\n"
        yaml += "semantic_height: " + str(src_rows) + "\n"
        yaml += "semantic_nodes: semantic.1.node\n"
        yaml += "slam_nodes: slam.1.node\n"
        yaml += "semantic_triangles: semantic.1.ele\n"
        yaml += "slam_triangles: slam.1.ele\n"
        f = open('registration.yaml', 'w')
        f.write(yaml)
        f.close()

        call(["mv", "registration.yaml", "register/"])
        call(["cp", self.img_2, "register/"])
        call(["cp", self.img_1, "register/"])
        filename = QFileDialog.getSaveFileName(self, 'Save As....', '', 'Map Registration Files
(*.mreg)')
        if not filename.endsWith(".mreg"):
            filename.append(".mreg")
        call(['mkdir', filename])
        call(['cp', '-r', 'register/', filename])

    # Triangulates both maps and writes these to file, along with the
    # colored triangle images
    def triangulate(self):
        self.savePoints()

        # Triangulate the nodes
        call(["./triangle/triangle", "./register/semantic.node"])
        call(["./triangle/triangle", "./register/slam.node"])

        # Build the triangulated, colored things
        self.color_triangles("slam")
        self.color_triangles("semantic")

        # Remove the extra .node files
        call(["rm", "register/semantic.node"])
        call(["rm", "register/slam.node"])

        self.robot.setTransforms(self.map1.get_points(), self.map2.get_points())
        self.toggleRobot.setEnabled(True)

    # Creates and writes the triangules based on the triangulation
    def color_triangles(self, image):
        if image == "semantic":
            node_file = "register/semantic.1.node"
            ele_file = "register/semantic.1.ele"
            rows, cols = cv2.imread(self.img_1, 0).shape
        elif image == "slam":
            node_file = "register/slam.1.node"
            ele_file = "register/slam.1.ele"
            rows, cols = cv2.imread(self.img_2, 0).shape
        else:
            return

        tri_img = np.zeros((rows, cols,3), np.uint8)
        # Seed nodes with None since triangles are 1-indexed
        nodes = self.nodes(node_file)
        # From the ele file, color triangles
    first_line = True
    with open(ele_file, 'r') as f:
        for line in f:
            if first_line:
                # Do nothing
                first_line = False
            elif '#' not in line:
                # This line is not a comment
                s = line.split()
                v1 = nodes[int(s[1])]
                v2 = nodes[int(s[2])]
                v3 = nodes[int(s[3])]
                pts = np.array([v1, v2, v3], np.int32)
                pts = pts.reshape((-1,1,2))
                color = self.robot.triangle_to_color(int(s[0]))
                cv2.fillPoly(tri_img,[pts],color)
    img_name = image + ".png"
    cv2.imwrite(img_name, tri_img)
    call(["mv", img_name, "register/" + img_name])

def outputWindow(self, image):
    cv2.imshow('Preview', image)
    # child = MyWindow(image, self)
    # child.show()

def export_zones(self):
    if self.export_ready:
        filename = QFileDialog.getSaveFileName(self, 'Save As...', '', 'YAML FILES (*.yaml)')
        if not filename.endsWith(".yaml"):
            filename.append(".yaml")
        fout = open(filename, 'w')
        # Do stuff to write to the file here
        converted_points = self.convert_points(self.myYaml)
        yaml.dump(converted_points, fout)
        fout.close()

def import_zones(self):
    fname = QFileDialog.getOpenFileName(self, 'Open File', "", 'YAML Files (*.yaml)')
    self.file = fname
    if fname.isEmpty():
        return
    self.export_ready = True
    fin = open(fname, 'r')
    with fin:
        # self.import_data = fin.read()
        self.myYaml = yaml.safe_load(fin)
        text = ""
        spacer = " \n"
    # For getting data from the YAML file know this
    #'Zone List' is a list of zones. For all of the zones, iterate through them
        # self.myYaml['ZoneList'][index]
    # Name is simply the name label as a string
    # Mode is an integer for the privacy type.
    # Points is another list of points, with X and Y values
    # Access points by using:
        # self.myYaml['Zone List'][index]['Points'][x/y]
        for i in range(0, len(self.myYaml['Zone List'])):
            text += self.myYaml['Zone List'][i]['Name'] + spacer
            text += self.privacyMode(self.myYaml['Zone List'][i]['Mode']) + spacer
            # Add the individual points
            for j in range(0, len(self.myYaml['Zone List'][i]['Points'])):
                text += str(self.myYaml['Zone List'][i]['Points'][j]) + spacer
            text += "-----\n"
        #Output to the preview window to make sure it's okay!
        self.zoneData.setText(str(text))
        #Import stuff from the yaml file here
    fin.close()
    self.show_zones_ckbox.setEnabled(True)

def convert_points(self, myYaml):
    dst = cv2.imread(self.img_2, 0)
    img_height, img_width = dst.shape

    for zone in myYaml['Zone List']:
        for point in zone['Points']:
            x = int(point['x'])
            y = int(point['y'])
            pt = self.robot.convert_to_2(QPoint(x, y))
            # Convert from slam image frame to the real world
            x = (pt[0] * self.slam_res) + self.slam_origin[0]
            y = self.slam_origin[1] - ((pt[1] - img_height) * self.slam_res)

            point['x'] = x
            point['y'] = y
    return myYaml

def privacyMode(self, mode):
    if mode == 1:
            return "Private"
        elif mode == 2:
            return "Public"
        else:
            return "No Filter"


class MyWindow(QtGui.QDialog):    # any super class is okay
    def __init__(self, image, parent=None):
        super(MyWindow, self).__init__(parent)
        self.export = QtGui.QPushButton('Export')

        self.pic = QtGui.QLabel()
        self.pic.setGeometry(10, 10, 100, 400)
        #use full ABSOLUTE path to the image, not relative
        self.pic.setPixmap(QtGui.QPixmap(image))

        layout = QtGui.QVBoxLayout()
        layout.addWidget(self.pic)
        layout.addWidget(self.export)
        self.setLayout(layout)
        self.export.clicked.connect(self.export_image)
    def export_image(self):
        pass

def main(argv):
    usage = "demo.py <Semantic Map Image> <SLAM Map Yaml>"
    src = ""
    dst = ""

    try:
        opts, args = getopt.getopt(argv, "h")
    except getopt.GetoptError as e:
        print usage
        sys.exit(2)

    if '-h' in opts:
        print usage
        sys.exit()

    if len(args) != 2:
        print usage
        sys.exit(2)
    src = args[0]
    dst = args[1]

    if src == "" and dst == "":
        print usage
        sys.exit(2)

    # Make the output directory
    call(["mkdir", "register/"])

    # Start ROS node
    rospy.init_node('map_registration_node')

    app = QApplication( sys.argv )
    mainWindow = MainWindow(src, dst)
    mainWindow.resize( 1000, 500 )
    mainWindow.show()
    app.exec_()

if __name__ == '__main__':
    main(sys.argv[1:])
```